Appendix A
Hands-on Experiments in GEZEL

A.1 Overview of the GEZEL Tools

GEZEL is a set of open-source tools for hardware/software codesign. The GEZEL website is at http://rijndael.ece.vt.edu/gezel2. This website distributes source code, pre-compiled versions (for Ubuntu), examples, and an online manual with installation instructions.

Figure A.1 shows an overview of the GEZEL tools. GEZEL is constructed as a C++ library with several components: a parser for the GEZEL language, a cycle-accurate simulation kernel, a VHDL code-generator, an interface to four different instruction-set simulators, and an interface to user-defined simulator extensions. The GEZEL tools are created on top of the GEZEL library. The examples throughout the book make use of these tools.

• fdlsim is the stand-alone cycle-accurate simulator. It uses the parser component, the cycle-accurate simulation kernel, and optional user-defined simulator extensions.
• gplatform is the co-simulator. It uses all of the components of fdlsim, in addition to several instruction-set simulators.
• fdlvhdl is the code-generator. It uses the parser component, the code-generator, and optional user-defined code-generation extensions.

For the reader interested in simulation tools, it is useful to study the user-defined simulation extension interface. This interface supports the creation of new ipblock types. All of the cosimulation environments were generated using this model.
A.2 Installing the GEZEL Tools

In this section, we briefly review the installation procedure for GEZEL. There are two different methods: installing pre-compiled packages, and recompiling from scratch. Installation of pre-compiled packages is the preferred method, in particular in a classroom environment. For example, the author has relied on the use of a Ubuntu Virtual Machine in his class, so that all students in the class can install a cosimulation environment on their own machine.

A.2.1 Installation on a Ubuntu System

Ubuntu uses the Debian packaging system. This provides an easy installation procedure. GEZEL is distributed as six packages. In the following package names, the dist suffix should be replaced with the name of the Ubuntu distribution. For example, on Ubuntu Precise (12.04), you would use gezel-base-precise.

- gezel-base-dist provides basic GEZEL capabilities, including simulation, cosimulation and code generation.
• **gezel-sources-dist** provides GEZEL source code modules, which you need if you want to recompile GEZEL from source.
• **gezel-debug-dist** provides a compiled version of the GEZEL tools with full debug info. This package is useful for GEZEL debugging purposes.
• **gezel-simulavr-dist** is a version of simulavr with GEZEL cosimulation interfaces. You need this package if you want to do AVR-based cosimulations.
• **gezel-simitarm-dist** is a version of Simit-ARM with GEZEL cosimulation interfaces. You need this package if you want to do ARM-based cosimulation.
• **gezel-examples-dist** contains demonstration examples of GEZEL, including many of the examples described in this book.

GEZEL packages are stored in a repository, a web server that provides easy access to the distribution. The URL of the GEZEL repository is [http://rijndael.ece.vt.edu/gezel2repo](http://rijndael.ece.vt.edu/gezel2repo). Installation of GEZEL packages now proceeds through four steps.

1. Configure the packaging system on your Ubuntu machine to read the GEZEL repository. The easiest way is to add the following line to the file `/etc/apt/sources.list`:
   ```
   deb http://rijndael.ece.vt.edu/gezel2repo precise main
   ```
   This line includes the name of the Ubuntu release. If you are working on Ubuntu Lucid (10.04), for example, you have to replace ‘precise’ with ‘lucid’. Also, you need superuser privileges to edit the file `/etc/apt/sources.list`. Use `sudo vi /etc/apt/sources.list`.

2. Add the author’s public key to your system. This will enable you to verify the authenticity of the packages. Adding a public key will require superuser privileges as well.
   ```
   sudo apt-key adv --keyserver pgp.mit.edu --recv-keys 092EF91B
   ```

3. Refresh the directory of available packages
   ```
   sudo apt-get update
   ```

4. Decide what GEZEL packages to install and proceed. For example, to install `gezel-base-precise` you would use
   ```
   sudo apt-get install gezel-base-precise
   ```
   Once you have configured the packaging system to recognize the GEZEL repository, you can easily remove and add packages with additional `apt-get` commands. Furthermore, if GEZEL is upgraded, you will automatically be notified by the Ubuntu system that an upgrade is available. The current version (Summer 2012) of GEZEL is 2.5.13. There are approximately two new releases of GEZEL per year.
The installation directory of the GEZEL tools is `/opt/gezel`. This directory is not included in the standard `PATH` of a Ubuntu system. To run, for example, `fdlsim` on the file `aes.fdl`, you should use the full path as follows.

```
/opt/gezel/bin/fdlsim aes.fdl 100
```

Alternately, you can adjust your path to include the GEZEL directory by default. This allows you to use just the GEZEL executable name in a command line.

```
export PATH=$PATH:/opt/gezel/bin
fdlsim aes.fdl 100
```

### A.2.2 Installation of Cross-Compiler Tools

The instruction-set simulators in GEZEL need a cross-compiler to generate binaries for simulation. Depending on the cores you wish to use, you need to install one of the following compiler toolchains.

- **Simit-ARM** uses an `arm-linux-gcc` compiler. The repository for GEZEL includes such a compiler (package `arm-linux-gcc`). Executables for Simit-ARM need to be created with the `-static` flag.
  ```
  /usr/local/arm/bin/arm-linux-gcc -static \
  -o myprogram \
  myprogram.c
  ```

- **Dalton 8051** uses the Small Devices C Compiler (package `sdcc`). `sdcc` is included in the ‘universe’ part of the Ubuntu Debian repository. Executables for the 8051 Dalton simulator are created as ihx files.
  ```
  sdcc myprogram.c
  ```

- **SimulAVR** uses the `avr-gcc` compiler (package `gcc-avr`). Additional packages include `binutils-avr` and `avr-libc`. `gcc-avr` is included in the ‘universe’ part of the Ubuntu Debian repository. Executables for SimulAVR are created as ELF files:
  ```
  avr-gcc -mmcu=atmega128 myprog.c -o myproc
  ```

### A.2.3 Compiling GEZEL from Source Code on a 32-bit System

Compiling GEZEL from source requires multiple compilation steps, and it requires additional packages on your system. The following steps illustrate compilation on a clean Ubuntu Precise (12.04) system.
Start by downloading the gezel-sources-precise package. You can either follow the package configuration steps above, or else directly download the package from the repository. Direct installation of a Debian package can be done with the `dpkg` command.

```
sudo dpkg -i gezel-sources-precise_2.5.13_i386.deb
```

To compile all source, you need to install several additional packages: `autoconf`, `libtool`, `g++`, `bison`, `texinfo`, `flex`, `libgmp3-dev`, `binutils-dev`. Use the following command.

```
sudo apt-get install autoconf libtool g++ bison flex texinfo libgmp3-dev binutils-dev
```

### A.2.3.1 Compiling the Stand-Alone Simulation Tool `fdlsim`

Extract the source code of the gezel simulation tools (`gplatform`, `fdlsim`)

```
tar zxfv /opt/gezel-sources/gezel-sim.tgz
```

```
cd gezel-sim
```

```
./bootstrap
```

To compile the stand-alone simulator, and install it in `gezel-sim/build`, use the following command:

```
./configure --enable-standalone
make install
```

### A.2.3.2 Compiling the Instruction-Set Simulator Simit-ARM

Before you can compile the GEZEL cosimulator, you will need to compile the instruction-set simulators you would like to use. The Dalton 8051 and Picoblaze ISS are integrated at source-code level. The Simit-ARM and simulavr simulators, however, are integrated at the library level. Compile Simit-ARM as follows. First, extract the source code of the instruction-set simulator.

```
tar zxfv /opt/gezel-sources/simit-arm-sfu.tgz
```
To compile simit-arm-sfu, and install it in /opt/simit-arm-sfu, use the following command. Note the use of the pre-processor directive CPPFLAGS, which is needed to enable cosimulation stubs in simit-arm.

```
cd simit-arm-sfu
./configure CPPFLAGS='-DCOSIM_STUB' \ 
   CXXFLAGS='-fpermissive' \ 
   --prefix=/opt/simit-arm-sfu
make
sudo make install
```

A.2.3.3  Compiling the Instruction-Set Simulator simulavr

First, extract the source code of the instruction-set simulator.

```
tar zxfv /opt/gezel-sources/simulavr.tgz
cd simulavr
./bootstrap
```

To compile simulavr, and install it in /opt/simulavr, use the following command.

```
./configure --prefix=/opt/simulavr
make
sudo make install
```

A.2.3.4  Compiling the Cosimulation Tool gplatform

To compile the cosimulator, and install it in gezel-sim/build, first install gezel-simitarm-dist and gezel-simulavr-dist, or compile them from source as indicated above. Then, use the following command:

```
./configure --enable-gplatform \ 
   --enable-simitarm --enable-simitsfu \ 
   --with-simit=/opt/simit-arm-sfu \ 
   --with-simulavr=/opt/simulavr
make install
```

A.2.3.5  Compiling the Code Generation Tool fdlvhd

Extract the source code of the gezel code generation tools (fdlvhd, igc)
A.2 Installing the GEZEL Tools

```
tar zxfv /opt/gezel-sources/gezel-cg.tgz
cd gezel-cg
./bootstrap
```

To compile the code generation tools, and install them in gezel-cg/build, use the following command:

```
./configure --enable-vhdl \ 
--enable-igc \ 
--with-gezel=/opt/gezel
make install
```

A.2.4 Compiling GEZEL from Source Code on a 64-bit System

The compilation for a 64-bit platform is largely identical to the compilation for a 32-bit platform. This section only explains the steps needed for 64-bit compilation.

The main difference between the 32-bit and 64-bit version of GEZEL is that the Simit-ARM instruction-set simulator is not available as part of the cosimulation.

To compile a 32-bit (standalone) version of Simit-ARM, first install multilib support for g++:

```
sudo apt-get install g++-multilib
```

A.2.4.1 Compiling the Instruction-Set Simulator Simitarm (on 64-bit Platform)

Extract the source code of the instruction-set simulator.

```
tar zxfv /opt/gezel-sources/simit-arm-sfu.tgz
```

To compile simit-arm-sfu, and install it in /opt/simit-arm-sfu, use the following command. Note the use of the pre-processor directive, which selects a 32-bit compile.

```
cd simit-arm-sfu
./configure CPPFLAGS=-m32 LDFLAGS=-m32 --prefix=/opt/simit-arm-sfu
make
sudo make install
```
A.2.4.2 Compiling the Cosimulation Tool gplatform (on 64-bit Platform)

To compile the cosimulator, and install it in `gezel-sim/build`, first install `gezel-simulavr`, or compile it from source as indicated above. Then, use the following command:

```
./configure --enable-gplatform \
    --with-simulavr=/opt/simulavr
make install
```

A.3 Running the Examples

The package `gezel-examples-dist` includes many examples from the book. Their source code can be found on `/opt/gezel-examples/bookex`. The examples are ready-to-run, assuming a complete GEZEL installation is available. This section briefly describes how to compile and run the examples.

A.3.1 Examples from FSMD Chapter

The `C05_FSMD` directory includes two subdirectories: `gcd` and `median`. The `gcd` directory contains the source code of a Greatest Common Divisor Algorithm in several Hardware Description Languages. The `median` directory includes three versions of the Median computation example: a version in C, a fully parallel hardware version, and a sequentialized hardware version. The following sequence of commands demonstrates their execution.

- Reference implementation in C.
  ```
  > make median
  gcc -o median median.c
  > ./median
  The median of 4, 56, 2, 10, 32 is 10
  ```

- Fully parallel version in GEZEL.
  ```
  > make sim1
  /opt/gezel/bin/fdlsim m1.fdl 1
  The median is 10
  ```

- Sequential version in GEZEL.
  ```
  > make sim2
  /opt/gezel/bin/fdlsim m2.fdl 200
  12 a1 1234/1234 q1 0
  ```
A.3 Running the Examples

A.3.2 Examples from Microprogrammed Architectures

Chapter

The C06_MICRO directory includes three examples: an implementation of the Hypothetical Microprogrammed Machine (HMM) in GEZEL, the Bresenham Algorithm written for an 8051 microcontroller, and the Bresenham Algorithm written for an 8051-based microprogrammed machine. The following sequence of commands demonstrates their execution.

• Hypothetical Microprogrammed Machine

    > make
    cpp -P hmm.fdl | /opt/gezel/bin/fdlsim 200
    0 IO 1 0 14 14
    1 IO 1 0 32 32
    2 IO 0 0 87 14
    3 IO 0 0 87 14
    4 IO 0 0 87 14
    5 IO 0 0 87 14
    6 IO 0 0 87 14
    7 IO 0 0 87 14
    8 IO 0 0 87 14
    9 IO 0 0 87 14
    ...

• Bresenham Algorithm on an 8051 Microcontroller

    > make
    sdcc bresen.c
    > make sim
• Bresenham Algorithm on an 8051-based Microprogrammed Machine

> make
sdcc bresen.c
> make sim
/opt/gezel/bin/gplatform bresen.fdl
i8051system: loading executable [bresen.ihx]
0 x 0/0 y 0/0 e 0/0 x2 0/0 y2 0/0 e2 0/0 xs 0/0 ...
0xFF 0xD7 0xFF 0xFF
0 x 0/0 y 0/0 e 0/0 x2 0/0 y2 0/0 e2 0/0 xs 0/0 ...
0x80 0xD7 0xFF 0xFF
80 x 0/0 y 0/0 e 0/0 x2 0/0 y2 0/0 e2 0/0 xs 0/0 ...
80 x 0/0 y 0/0 e 0/0 x2 0/0 y2 0/0 e2 0/0 xs 0/0 ...
0x00 0xD7 0xFF 0xFF
0 x 0/0 y 0/0 e 0/0 x2 0/0 y2 0/0 e2 0/0 xs 0/0 ...
0x00 0x17 0xFF 0xFF
...

A.3.3 Examples from System on Chip Chapter

The C08_SOC directory includes two examples. The first, pingpong, illustrates a pingpong buffer communication scheme between an 8051 and hardware. The second, uart, shows simple UART analyzer (written in GEZEL), attached to an AVR microcontroller. The following sequence of commands demonstrates their execution.

• Ping Pong buffer in 8051

> make
sdcc ramrw.c
> make sim
/opt/gezel/bin/gplatform -c 50000 pingpong.fdl
i8051system: loading executable [ramrw.ihx]
0x00 0x00 0xFF 0xFF
A.3 Running the Examples

0x01 0x00 0xFF 0xFF
28984 ram radr 0/1 data 40
28985 ram radr 1/2 data 3f
28986 ram radr 2/3 data 3e
28987 ram radr 3/4 data 3d
28988 ram radr 4/5 data 3c
0x00 0x01 0xFF 0xFF
38464 ram radr 20/21 data df
38465 ram radr 21/22 data de
38466 ram radr 22/23 data dd
38467 ram radr 23/24 data dc
38468 ram radr 24/25 data db
0x01 0x00 0xFF 0xFF
47944 ram radr 0/1 data 80
47945 ram radr 1/2 data 7f
47946 ram radr 2/3 data 7e
47947 ram radr 3/4 data 7d
47948 ram radr 4/5 data 7c
Total Cycles: 50000

• AVR UART Analyzer

> make
avr-gcc -mmcu=atmega128 avruart.c -o avruart.elf
> make sim
/opt/gezel/bin/gplatform -c 8000 uart.fdl
atm128core: Load program avruart.elf
atm128core: Set clock frequency 8 MHz
@237: ->1
@361: ->0
@569: ->1
@1401: ->0
@1609: ->1
@1817: ->0
@2233: ->1
@2649: ->0
@2857: ->1
@3689: ->0
@3897: ->1
@4105: ->0
@4521: ->1
Total Cycles: 8000
A.3.4 Examples from Microprocessor Interfaces Chapter

The C11_ITF directory includes two examples. The first, \texttt{gcd}, demonstrates a memory-mapped GCD algorithm. The second, \texttt{endianess}, demonstrates four different implementations of an Endianess conversion module. The four implementations are a software implementation, a memory-mapped coprocessor, and two different ASIP implementations. The following sequence of commands demonstrates their execution.

- Memory-mapped Greatest Common Divisor

  
  ```
  > make
  /usr/local/arm/bin/arm-linux-gcc -static \gcddrive.c \-o gcddrive
  > make sim
  gplatform gcdmm.fdl
core my_arm
armsystem: loading executable [gcddrive]
armsystemsink: set address 2147483652
armsystemsink: set address 2147483660
gcd(80,12) = 4
gcd(80,13) = 1
Total Cycles: 14764
  ```

- Endianness Conversion: all software design

  ```
  > make
  /usr/local/arm/bin/arm-linux-gcc -O3 \-static \-o endian.elf \endian.c cycle.s
  > make sim
  /opt/gezel/bin/gplatform endian.fdl
core myarm
armsystem: loading executable [endian.elf]
4K conversions take 53786 cycles
   Per conversion: 13 cycles
Total Cycles: 70035
  ```

- Endianness Conversion: memory-mapped coprocessor design

  ```
  > make
  /usr/local/arm/bin/arm-linux-gcc -O3 \-static \-o endian.elf \endian.c cycle.s
  ```
A.3 Running the Examples

> make sim
/opt/gezel/bin/gplatform endian.fdl
  core myarm
armsystem: loading executable [endian.elf]
armsystemsink: set address 2147483652
4K conversions take 41502 cycles
  Per conversion: 10 cycles
Total Cycles: 57746

- Endianess Conversion: ASIP design with single-argument instructions

> make
/usr/local/arm/bin/arm-linux-gcc -O3 -static \
  -o endian.elf \
  endian.c cycle.s

> make sim
/opt/gezel/bin/gplatform endian.fdl
  core myarm
armsystem: loading executable [endian.elf]
4K conversions take 37401 cycles
  Per conversion: 9 cycles
Total Cycles: 57702

- Endianess Conversion: ASIP design with double-argument instruction

> make
/usr/local/arm/bin/arm-linux-gcc -O3 -static \
  -o endian.elf \
  endian.c cycle.s

> make sim
/opt/gezel/bin/gplatform endian.fdl
  core myarm
armsystem: loading executable [endian.elf]
4K conversions take 29209 cycles
  Per conversion: 7 cycles
Total Cycles: 47459

A.3.5 Examples from Trivium Chapter

The C13-Trivium directory includes four examples, all of them based on the Trivium coprocessor. The first, trivium_hw, includes three different implementations of the Trivium design as a stand-alone module. The second, trivium_arm_sw, shows a reference implementation of Trivium in software, for ARM. The third,
trivium_arm_t32 demonstrates a memory-mapped coprocessor for ARM. The last one, trivium_arm_sfu shows a custom-instruction design of Trivium, emulated on an ARM. The following sequence of commands demonstrates their execution.

- **Trivium standalone module.** Use `make sim1`, `make sim2` or `make sim3` to run a 1 bit-per-cycle, 8-bit-per-cycle or 32-bit-per-cycle implementation. The output of the 32-bit-per-cycle simulation is shown below.

```
> make sim3
/opt/gezel/bin/fdlsim trivium32.fdl 50
39 11001100110011100111001110101111011101001000000 99bd7920
40 10011001101110101101101101010010001000 9a235a88
42 0001001001010001000011111111101001011111 1251fc9f
43 101011111111100010100110010101011010101 aff0a655
44 011111101100100011101110010011100 9a235a88
45 1011111111110101000010000100100010001000 7ec8ee4e
46 1000011010101011100110010100010000100101000 bdf42128
47 1000000000110111010100111001011111111010110 86dae608
48 010110001011011010111000010000100001000 1251fc9f
49 00010110111110101000100011110100 16fa88f4
```

- **Trivium reference implementation in software**

```
> make
/usr/local/arm/bin/arm-linux-gcc -static \
-O3 \
trivium.c cycle.s \
-o trivium
> make sim
/opt/gezel/bin/gplatform trivium32.fdl
core myarm
armsystem: loading executable [trivium]
7b
75
ce
cc
...
51
12
key schedule cycles: 3810 stream cycles: 48815
Total Cycles: 81953
```

- **Trivium memory-mapped coprocessor**

```
> make
/usr/local/arm/bin/arm-linux-gcc -static \
```
A.3 Running the Examples

trivium.c cycle.s\  
-o trivium

> make sim
/opt/gezel/bin/gplatform trivium32.fdl
core myarm
armsystem: loading executable [trivium]
armsystemsink: set address 2147483648
armsystemsink: set address 2147483656
ccce757b ccce757b 99bd7920 9a235a88 ... 
86dae608 806ea7eb 58aec102 16fa88f4 ... 
... c2cecf02 c18e5cbc 533dbb8f 4faf90ef ...
key schedule cycles: 435 stream cycles: 10524
Total Cycles: 269120

- Trivium custom-instruction design

> make
/usr/local/arm/bin/arm-linux-gcc -static \  
  trivium.c cycle.s\  
  -o trivium

> make sim
/opt/gezel/bin/gplatform triviumsfu.fdl
core myarm
armsystem: loading executable [trivium]
ccce757b 99bd7920 9a235a88 1251fc9f ... 
806ea7eb 58aec102 16fa88f4 c5c3aa3e ... 
key schedule cycles: 289 stream cycles: 8862
Total Cycles: 39219

A.3.6 Examples from AES Chapter

The C14_AES subdirectory includes the design of a memory-mapped coprocessor for ARM. The following sequence of commands demonstrates its execution.

- AES memory-mapped coprocessor

  > make
  /usr/local/arm/bin/arm-linux-gcc -static \  
    aes_coproc_armdriver.c \  
    -o aes_coproc_armdriver

  > make sim
  /opt/gezel/bin/gplatform aes_coproc_arm.fdl
  core myarm
  armsystem: loading executable [aes_coproc_armdriver]
armsystemsink: set address 2147483652
cycle 10164: set key0/1020305060708090a0b0c0d0e0f
cycle 10222: set text_in 0/112233445566778899aabbcc
ddeeff
cycle 10235: start encryption
cycle 10255: get text_out 69c4e0d86a7b0430d8cdb78070b4c55...
text_out 69c4e0d8 6a7b0430 d8cdb780 70b4c55a
Total Cycles: 15188

A.3.7 Examples from CORDIC Chapter

The C15_CORDIC subdirectory includes the design of an FSL-mapped CORDIC design. The FSL link is emulated on an ARM processor, as explained in Chap. 15. The following sequence of commands demonstrates its execution.

- CORDIC FSL-mapped accelerator
  ```
  > make
  /usr/local/arm/bin/arm-linux-gcc -static
  -O3 \
  cordic.c \ 
  -o cordic
  > make sim
  /opt/gezel/bin/gplatform cordic.fdl
  core arm1
  armsystem: loading executable [cordic]
  Checksum SW 55affcee FSL 55affcee
  Total Cycles: 23228
  ```
References


References

Micheli GD, Benini L (2006) Networks on chips: technology and tools (Systems on silicon). Morgan Kaufmann, San Francisco
Muchnick SS (1997) Advanced compiler design and implementation. Morgan Kaufmann, San Francisco
Qin W, Malik S (2003) Flexible and formal modeling of microprocessors with application to retargetable simulation. In: DATE ’03: proceedings of the conference on design, automation and test in Europe, Munich, p 10556


Vahid F (2007b) It’s time to stop calling circuits “hardware”. Computer 40(9):106–108


Index

Numbers
8051 in GEZEL, 254

A
address decoding, 289
admissible schedule, 39
Advanced Encryption Standard, 409
Ahmdahl’s law, 24
Application Binary Interface, 220
ASIC, 18
ASIP, 12, 18
assembler, 194

B
bandwidth
   off-chip, 242
   on-chip, 242
big-endian, 206
bit-parallel processor, 281
bit-serial processor, 281
block cipher, 375
blocking, 277
boolean data flow, 45
bus
   alignment, 296
   burst transfer, 299
   clock cycle, 292
   locking, 305
   multi-layer, 307
   naming convention, 291
   overlapping arbitration, 303
   pipelined transfer, 299
   split transfer, 299
timeout, 294
timing diagram, 292
topology, 307
transfer sizing, 296
wait state, 294
bus arbiter, 288, 302
bus bridge, 238
bus master, 238, 288
bus slave, 238, 288

C
cache
   set-associative, 226
cast
   in GEZEL, 116
CFG, see control flow graph
ciphertext, 375
circular queue, 63
code inlining, 76
communication-constrained coprocessor, 280
compiler, 194
computation-constrained coprocessor, 280
computational efficiency
   actual, 242
   intrinsic, 241
concurrency, 24
Connection Machine, 24
continuous-time model, 22
control edge, 90
control edge implementation, 92
control flow graph, 90, 93
   construction from C, 95
   for control statements, 94
   of euclid’s algorithm, 94
control flow modeling, 45
control hierarchy, 359
control path, 95
control processing, 21
control store, 159
  control store address register, 159
cooperative multi-threading, 71
coprocessor argument, 353
coprocessor interface, 328
coprocessor parameter, 353
CORDIC, 435
  rotation mode, 454
  vector mode, 454
critical path, 77
crossbar, 307
CSAR, see control store address register
custom-instruction interface, 334
cycle-accurate model, 23
cycle-based hardware, 3

D
data edge, 90
  in control statements, 91
data edge implementation, 92
data flow
  actor, 36
  actor implementation, 64
  firing rule, 37
  marking, 37
  multi-rate, 51
  queue, 36
  token, 35
data flow graph, 90, 95
  construction from C, 96
data flow graphs, dealing with pointers and arrays, 97
data flow model, 34
data path, 100
data processing, 21
data-stationary control, 362
dataflow, interleaving, 75
deadlock, 39
determinate specification, 38
DFG, see data flow graph
dis-assembling, 209
discrete-event model, 23
distributed
  communication (in SoC), 242
data processing (in SoC), 241
  storage (in SoC), 243
DM310 processor, 247
domain-specific, 18
DRAM, 244
DSP, 12, 18

E
Efficiency
  energy, 19
  time, 19
endianness, 336
energy efficiency, 15
energy-efficiency, 13
Euclid’s algorithm, 77
expression
  in GEZEL, 118

F
fast simplex link, 330, 448
FIFO, 63
finite state machine, 101, 122
  in GEZEL, 126
  Moore and Mealy, 124
finite state machine with datapath, 125
  execution model, 129
  implementation, 132
  language mapping, 143
  limitations, 157
  modeling trade-off, 130
  proper FSMD, 141
firing vector, 41
fixed point representation, 437
Flexibility, 18
flexibility, 20
FPGA, 11, 18
FSM, see finite state machine
FSMD, see finite state machine with datapath

G
GEZEL
  code generator, 447

H
handshake, 275
hardware interface, 353
  index-multiplexing, 356
  time-multiplexing, 356
hardware interface port, 355
hardware sharing factor, 280
hardware-software codesign
  definition, 11, 12
heterogeneous
  communications (in SoC), 242
  data processing (in SoC), 241
  storage (in SoC), 243
hierarchical control
in SoC, 247

I
instruction
in GEZEL, 126
instruction-accurate model, 23
instruction-set simulator, 223
interface
coprocessor, 240
processor custom-datapath, 240
SoC peripheral, 239
IP reuse, 21
ipblock, 324
  in GEZEL, 251

K
key schedule, 375
keystream, 375

L
linear feedback shift register, 119
linear pipeline, 332, 361
linker, 194
little-endian, 206
loader, 194
loose-coupling, 282

M
mailbox, 320
master handshake, 322
memory
  access time, 244
  cell size, 244
  power consumption, 245
  retention time, 244
memory wall, 245
memory-mapped coprocessor, 324
memory-mapped register, 317
methodology, 19
micro-instruction
  formation, 166
  micro-instruction encoding, 161
  horizontal encoding, 162
  vertical encoding, 162
micro-program interpreter, 175
  macro-machine, 177
  micro-machine, 177
micro-program pipelining
  control store output pipeline, 181
  CSAR update loop pipeline, 182
  datapath condition register, 181
  multi-rate dataflow graph, 38

N
network on chip, 307
non-blocking, 277
non-linear pipeline, 332, 361
NVRAM, 244
NVROM, 244

O
one-way handshake, 275
operator compounding, 343
operator fusion, 343
operators
  in GEZEL, 118

P
parallelism, 24
PASS, see periodic admissible schedule
periodic admissible schedule, 40
plaintext, 375
platform, 17
port-mapped interface, 383
producer/consumer, 275
programmable, 16

R
rank of a matrix, 41
reconfigurable, 16
register
  in GEZEL, 113
  in hardware, 243
reservation table, 363
RISC, 17, 199
  control hazard, 201
  data hazard, 202
  data type alignment, 205
  delayed-branch, 201
  interlock, 200
  link register, 210
  pipeline hazard, 200
  pipeline stall, 200
  scalability, 194
  structural hazard, 204
round-robin, 72
RTL, 3
S
scheduler, 71
SDF, 38
shared memory, 323
simulation, 15
single-assignment program, 103
merge function, 104
single-thread software, 6
slave handshake, 322
SoC
platform, 237
soft-core, 11
spatial decomposition, 20
SRAM, 244
state explosion, 157
static schedule, 62, 75
stream cipher, 375
StrongARM
in GEZEL, 251
structural hierarchy, 121
synchronization dimensions, 271
synchronization point, 271
synchronous dataflow graph, 38
systolic-array processor, 281
T
tight-coupling, 282
time-stationary control, 362
timewise decomposition, 20
topology matrix, 40
transaction-accurate model, 23
Trivium, 375
two-way handshake, 275
V
volatile pointer, 318
W
wire
in GEZEL, 114
Y
yield point, 71